Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/8140.2--96-7818

Lossless Data Compression of Packet Data Streams

JUNHO CHOI

Command Control Computers and Intelligence Branch
Space Systems Development Department

MITCHELL R. GRUNES

Allied Signal Technical Services
Camp Springs, MD

February 14, 1996

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 14, 1996	

4. TITLE AND SUBTITLE

Lossless Data Compression of Packet Data Streams

5. FUNDING NUMBERS

PE - 81-M040-X6

6. AUTHOR(S)

Junho Choi, Mitchell R. Grunes*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/8140.2--96-7818

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

SPARWAR

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*Allied Signal Technical Services, 5801 Allentown Rd Suite 400, Camp Springs MD 29746

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This report continues the work published in the prior interim report, NRL/MR/8140.2-95-7742. It presents the interim development of a compression and de-compression system, including tests of effectiveness on selected digitized packet streams. Blocks of packets are grouped by packet type, and are broken up into separate compression streams on bit field boundaries. 18 lossless compression algorithms are examined for effectiveness. These algorithms are able to compress most of the data available with a good compression factor, except on encrypted data.

14. SUBJECT TERMS

Arithmetic coding	LZ77	LZRW3A	Radix coding
Data compression	LZ78	Packet	Rice coding
Lossless compression	LZW	PPMC	Run length encoding

15. NUMBER OF PAGES

51

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

TABLE OF CONTENTS

LOSSLESS DATA COMPRESSION OF PACKET DATA STREAMS

1. Introduction

1.1 Background

This report continues the work published in the prior interim report, reference [3]. It presents the current development of a compression and de-compression prototype system and tests of effectiveness on the selected digitized packet streams. It is also an expanded version of a previously published conference paper [4]. It is augmented by the description of data formats contained in [2].

We have created a proof of feasibility prototype system for compressing packet data streams. The end goal is to reduce costs. Real world constraints include transmission in the presence of error, tradeoffs between the costs of compression and the costs of transmission and storage, and imperfect knowledge of the data streams to be transmitted. The overall method is to bring together packets of similar type, split the data into bit fields, and test a large number of compression algorithms. The best algorithm is chosen for each field.

Results are very promising, frequently offering compression factors substantially higher than those obtained with generic byte stream compressors, such as Unix Compress and HA 0.98. Dramatic improvements over the 1994 fiscal year (FY 94) have occurred as result of two major factors: more and improved compression algorithms, and better knowledge of the data formats.

The creation of complex technical systems in the real world environment often involves a major administrative effort to integrate the equipment and data created by diverse organizations. Each organization typically brings its own interests and prior experience, builds its own instruments, and defines its own data stream content.

An administratively efficient solution to data stream integration is to organize the data into "packets" containing a common set of basic bit fields, and one or more variable format fields. Each organization is free to define the structure and content of the variable format fields for each of its packet types. The common elements allow hardware and software to deal with all types of data in a similar manner, at the source and at the destination.

Manuscript approved January 5, 1996.

A simple example might include:

Field	Bits	Content
		-----COMMON FIELDS-----
1	4	Organization defining this packet particular type.
2	5	Packet type within that organization.
3	3	Telemetry flags indicating overall equipment status.
4	28	Time tag for packet generation.
5	32	-----VARIABLE FORMAT FIELD----- Subdivided differently for each packet type

One packet structure might define the variable format field to contain four 8 bit values, all part of the same data sequence. Another packet structure might contain three fields, with 9, 11 and 12 bits, in distinct data sequences.

The software is currently configured to be used with a set of somewhat more complex packet formats outlined in [2].

The administrative efficiency is obtained at the cost of some increase in data bandwidth. In addition, some packet telemetry fields change rather slowly. Thus, a data compression system which has been customized to the specific packet structures has herein been found to attain lossless compression factors that are higher than those obtained in typical consumer applications.

1.2 Objectives

The main objective of this study has been to develop a compression algorithm that can generate substantial cost savings and bandwidth reduction. The work has included the development of a proof of feasibility prototype compression and de-compression system.

Data transmission and storage in modern remote sensing systems represent major components of the total cost. In particular, data volume affects bandwidth, transmit power, the cost and payload weight of power systems, the cost of transmission lines and channels, and storage requirements. These concerns apply both on-board and on the ground.

For some applications inexpensive off-the-shelf compressors provide adequate results. This study is directed towards those applications where the high costs of transmission and storage justify a more elaborate system. This must always be balanced against the cost and power requirements of the data compression system itself.

1.3 System Design Considerations

Our present system can be customized for compression and de-compression in multiple applications by modifying those portions and tables which define and manipulate packet types, or which input or output data. Flexibility is important, since one may have imperfect knowledge about the format and statistics of the input data stream at the design stage of the communications system, when many design elements are still in flux.

Compressed data is more context-dependent than uncompressed data, so error correction coding (ECC) is very strongly recommended for channels that transmit compressed data. It is assumed that one uses external ECC hardware or software that is off-the-shelf or embedded in the data channel. This may mean that the compression system will not be able to determine when and where the ECC has lost badly transmitted bits. Therefore some overhead has been included in the compressed data stream to detect errors and to re-sync the data. A block of packets can be lost when an uncorrectable error occurs, but the rest of the data stream can still be recovered. This capability is extremely important.

The current compression software includes the error detection overhead, but the de-compression software is not yet capable of using it to recover from errors.

Other complications of the real-world exist. There is a great deal of externally produced freely available compression source code, although one must still deal with patents (see [5]). Indeed, if one is to believe [5], the highest compression factor generic byte stream compressors are freely available. The present system includes two such algorithms. One (LZRW3A) was of simple structure and was designed in such a way as to be externally callable; it was integrated with little difficulty. The other (HA 0.98) was a very complex stand-alone package; it presented greater difficulties.

A significant effort is required to embed externally produced stand-alone software inside any transmission system. In an operational environment one must ensure that no component can accidentally or deliberately halt or do harm to associated computers, networks and transmission channels. This involves a great deal of work, in terms of comprehending, re-writing, and simplifying source code. For our system we modified the interface structure, added some bounds checking for indices and pointers, and checked for other potential problems. Finally, in one of the cases (HA 0.98) it was considered prudent to check the de-compressed result before choosing the algorithm as best.

We impose the formal requirement for the lossless algorithm that the compression/de-compression process will cause no data to be lost or re-ordered, even if it does not match the assumed format. That is, the input stream of the compression process must be bit-for-bit identical to the output stream of the de-compression process. This is necessary to simplify debugging in complex environments where multiple organizations are involved.

The current version can also be configured to utilize a simple lossy compression algorithm which could further improve compression factors by omitting or reducing the precision of some fields (packet order is still preserved). It is expected that the decision to employ lossy compression would involve a careful cost-benefit analysis by each data user organization and project. It is recommended that the decision to employ lossy compression not be imposed on the data users. All of the results cited in this report are for lossless compression, unless otherwise specified.

Prior to compression, packets are classified into types. The less common types are classified together into one generic type, and packets not matching an expected format are classified into a byte stream type. The generic type breaks out the common format fields, but views the variable format fields as a sequence of bytes. The byte stream packet type views the entire packet as a sequence of bytes. Types with a small number of packets are re-classified as generic, and packets surrounded by byte stream packets are themselves re-classified as byte stream packets.

The runs of each packet type may not be very long, so it would involve too much bit overhead to start a new run of compression processing each time the packet type changes. Therefore, the data is divided into 512 packet blocks, packets are classified by type (as specified above), and the sequence of packet types is itself compressed and transmitted. All of the packets of each given packet type from a given

block are assembled into a separate compression sub-stream. On de-compression, the packet type sequence is used to re-assemble the packets back into the original order.

This blocking introduces a delay of at least 512 packets before information can be transmitted, a serious problem on temporarily quiescent data streams. To solve that problem, the block would be terminated after less than 512 packets, if the packet buffer does not fill quickly enough (in the operational version).

Each of the bit fields for each sub-stream is compressed as an independent compression sequence. In the current version, each bit field sequence is tested against 18 compression algorithms, and the best is chosen.

The output compressed data packet output from each block of input packets includes:

Sync code
Number of original packets (usually 512)
Compression algorithm # for type sequence
Compressed packet type sequence
For each packet type:
    For each bit field:
        Compression algorithm #
        Compressed data sequence
Check Sum

In a future version the output format should also include a configuration version number. This will allow multiple configurations to be in simultaneous use by different groups, or for different purposes. For example, some configurations might allow lossy compression (and attendant loss of precision) on some fields.

On reception, an incorrect sync code or check sum indicates that a compressed packet has been transmitted incorrectly. The intended mode of operation for the proposed system is that if either is wrong (or the error correction system does in fact inform the compression system of the location of an error), or another error occurs during de-compression, the block is discarded, and the receiver scans for the next sync code. Good blocks are de-compressed, and the packets are placed back into the original order. In the current prototype, errors merely abort the program.

2. Data Statistics Which Favor Compression

Good compression results can have significant economic value and can significantly impact professional stature. As, in any field, some claims tend to show a given system in the best possible light. For example, some forms of compression have been labelled "lossless" which do produce de-compressed data streams which are not bit-for-bit identical to the original streams. In most cases terms like

no observable loss
no perceivable loss
visually lossless

have been used to indicate that such a special definition is being employed. Our lossless compression algorithm is genuinely bit-for-bit lossless.

The system described herein is not capable of losslessly compressing all possible data sets. That is impossible, as shown by the well known "counting argument": There are 2^N possible data sets containing N bits. 2^N is monotonically increasing. Therefore it is impossible to represent all data sets by shorter ones. For example, one cannot uniquely represent all four possible 2-bit data sets by the two possible 1-

bit data sets. In order to compress *any* data set it is necessary (when all the overhead is taken into account) to represent some data sets using "compressed" data sets which are slightly larger than the originals.

Thus, a statement that some lossless compression system can compress all data sets by a given factor (or at all) would actually be an assumption that all data sets which actually occur will follow statistics favorable to that compression system--an optimistic assumption.

Lossless data compression functions by representing more probable data sets (those with favorable statistics) in fewer bits than the corresponding original data sets, at the expense of representing less probable data sets in more bits. If one knows enough about the format and statistics of a class of non-random data sets, one can do so in such a way that the average length of the "compressed" data sets will be less than the average length of the original data sets.

For additional information on this topic, consult [5].

The compression system described is only intended to compress data of the packet structures that it is configured to compress, on the assumption that some fields have statistics which are favorable for compression, namely:

The values or their first differences vary smoothly or slowly.
or Some values or their first differences occur more often than others.
or Some short sequences of values or their first differences occur more often than others.
or The values or their first differences do not fill the full dynamic range allowed by the number of bits.

Data whose statistics are approximately random (such as encrypted data, or previously compressed data streams) can generally not be losslessly compressed by this or any system.

Since no lossless compression factor can be assured with absolute certainty, operational systems transmitting on fixed data rate channels must be able to continue after some data is lost due to an insufficient compression factor. If an adequate safety margin is employed, such an occurrence will be rare.

3. Lossless Compression Algorithms

No radically new compression algorithms have been developed for this study, but some improvements have been made to published algorithms, and some published algorithms have been combined into hybrids. Our full algorithm can be considered to be a hybrid of all of the discussed algorithms.

Each algorithm is tried on each field, of each packet type sub-stream, within each block of packets. Current lossless algorithms include:

1. Constant Coding--If all of the values are of one constant value, that value is sent only once, and no other algorithms need be tested.

2. Constant Bit Removal--If only some of the bits are constant, a mask is sent specifying which bits are constant, and those bits are sent only once. *Constant bit removal is applied before all of the remaining algorithms.* For method 2, the remaining bits are transmitted unchanged.

3. Run Length Encoding--The sequence of values is replaced by a set of ordered pairs, containing the value, and the number of times it is consecutively repeated (less one). If the value changes infrequently, this will require fewer bits than the original.

   As an improvement to the algorithm, the number of bits needed to code the largest repeat count (less one) is determined and before transmitting the run-length pairs. That value and the number of values remaining are used to determine how many bits are used to send each run length code.

4. Rice Coding--This algorithm, based on [7] and [8], allows for very rapid adaptivity, because it transmits one adaptive parameter--the number of least significant bits of the remapped differences that may optimally be transmitted unchanged--for each Rice block (currently 32 values; could be varied). The most significant bits are re-mapped and transmitted as a terminated unary code (0 as 1, 1 as 01, 2 as 001..., with no termination required for the largest detected value). The differencing and remapping algorithms are discussed in the next section.

   A fairly large amount of work went into making improvements to this algorithm. For example, two special adaptive parameter values handle the low entropy cases: one indicates that all of the differences are zero, and the other adds one coding value to represent four zero difference values. The adaptive codes are themselves Rice coded within each block of packets.

5. LZ77--This was based on [9]. It is a "dictionary search" algorithm intended to be used with data having repeated strings of values. A window of prior values is searched for the longest string of values matching the current and future values. The backwards distance of the best match (or a flag value for new values not matching prior values) is transmitted, as is the number of matching values in the string. Flag values are prefixed before transmittal of values not previously sent.

   Improvements were made to the algorithm: an elaborate adaptive technique is used to determine the adaptive window size, based on prior matches, with a periodically increased size. A similar adaptive technique is used to determine the maximum string length. These sizes are rounded up to fit into an integral number of bits, and each value is transmitted in the minimum number of bits. Linked lists are used to speed up processing.

6. LZ77 Applied to Differences--The above algorithm is applied to the remapped differences.

7. LZRW3A--This LZW family algorithm, based on [11], was implemented in a software package that was obtained freely. The hash table depth was increased from 3 to 6. The software structure was quite simple, and no problems were encountered embedding it into our system. It was designed to deal with byte streams, so the field is copied into a byte stream: fields with 1-8 bits/value (after constant bit removal) are copied to one byte/value; values with 2-16 bits/value are copied to two bytes/value, etc.

8. LZRW3A Applied to Differences--The above algorithm is applied to the remapped differences.

9. LZ78--Another LZW dictionary search algorithm, based on [10], searches for groups of past strings. The same improvements in adaptive window sizing and the use of linked lists are applied as for LZ77.

10. LZ78 Applied to Differences--The above algorithm is applied to the remapped differences.

11. HA 0.98--This is file archiver that was rated by [5] as providing the highest compression factors on generic data sets, using an "improved" PPMC (Prediction by Partial Matching-C) algorithm with 4th order Markov modeling. The modeling is used to form probability weights for arithmetic coding. As with LZRW3A, the data is placed into a byte stream. HA 0.98 actually includes two somewhat different algorithms--"ASC" and "HSC". Both are tested.

    The HA 0.98 software was originally designed as a very complex stand-alone program to perform actions in response to command strings, and to interact with the operating system via elaborate system calls. Safely embedding this complex stand-alone program into our application consumed substantial time and effort, and the software runs somewhat slowly, but it does yield excellent compression factors on many fields, in many cases.

12. HA 0.98 Applied to Differences--The above algorithm is applied to the remapped differences.

13. Radix Coding--The sequence of reduced values are interpreted as the base M+1 representation of a single large number, where M is the maximum possible value. The large number is then transmitted.

    As improvements, the minimum value is found and subtracted from all data values, and M is set to the actual maximum residual. The large value accumulation is flushed when the maximum accumulation becomes too large for efficient calculation (currently, when it exceeds 2^{800}).

    We tried to use mixed radix (M) coding in hybrid with other algorithms (e.g., to code a fractional number of bits in a modified Rice algorithm), but those hybrids were not found to represent significant improvements and were dropped.

14. Radix Coding Applied to Differences--The above algorithm is applied to the remapped differences.

15. Arithmetic Coding--This algorithm is based on [1]. It is based on a simple zero-order incrementally adaptive probability model.

16. Arithmetic Coding Applied to Differences--The above algorithm is applied to the remapped differences.

17. Run Length Encoding + Rice--The run length pairs are derived as for algorithm 3. The values are then differenced, remapped and Rice coded. Remapping is modified to take advantage of the fact that values are never repeated. The run lengths are also Rice coded.

Our lossless algorithm is a hybrid of all of the above algorithms. As discussed earlier, packets are grouped together by type, the data is broken up into bit fields, and the algorithm is chosen for each field that produces the highest compression factor. In the event that no algorithm leads to an improvement, the field is sent uncompressed. A code indicating which algorithm was used for each field precedes the rest of the compressed stream.

4. Adaptive Differencing and Remapping

Many of the compression algorithms that we use rely on differencing of the values from their predicted values to reduce the typical size of the numbers to be coded. A variety of methods is tried.

In the simplest methods, the prediction for any value is simply the prior value. The differences of the current value from its predicted value are then remapped into a sequence of non-negative numbers. The

optimal way to perform this remapping depends on the proper interpretation of the bit field. All of the following interpretations are tested, to minimize the sum of the remapped values:

(A) No remapping (or differencing).
(B) The values are the least significant portion of larger values; positive differences more likely.
(C) Same as B, negative differences more likely.
(D) The values are unsigned; positive differences more likely.
(E) Same as D, negative differences more likely.
(F) The values are signed, in two's complement notation; positive differences more likely.
(G) Same as F, negative differences more likely.

For example, the sequence
5,5,4,5,3,5,2,5...
would produce the difference sequence
0,-1,1,-2,2,-3,3...
which would be remapped using assumption (C) as
0,1,2,3,4,5,6...

The greatest common factor is removed at two points in the process, and minimum values are found and subtracted. If no negative or no positive differences occur and/or there is a minimum absolute difference, those facts are also taken advantage of. The largest values are found at two points in the algorithm, and are used to determine the number of bits needed to transmit the remapped values.

Finally, a least squares fit is tried in which the current difference is predicted to be
alpha+beta*(previous difference)
The fit is only used when it improves matters. It often does not, because residuals have fewer removable systematic patterns than the differences themselves, and because of overhead.

The current algorithm treats the data as one dimensional sequences. A two dimensional predictor would be desirable if this compression system were applied to images.

5. Lossy Compression Algorithm

In the event that the software is configured to apply lossy compression, the software currently employs a very simple algorithm, which reduces the precision of the values sent.

The configuration for each field within each packet type includes a value called iLoss. If iLoss is zero (or is larger than the largest possible value), the field is not sent, and will be de-compressed as zero. If iLoss is one, the field is sent using lossless compression. If iLoss is greater than one, the field values are divided by iLoss and rounded down. It is then sent using the hybrid lossless compression algorithm discussed in the prior sections. After the field is de-compressed, it is multiplied by iLoss, then added to iLoss/2 (rounded down). That means that the maximum possible de-compression error is iLoss/2 (rounded down).

This form of lossy compression is in some respects sub-optimal. It was chosen at the current time for two reasons:

A. It is expected that those data users considering the use of lossy compression will do so with some trepidation. They may only consider methods whose affects they can easily and completely understand.

B. Those data users should have the option of selectively transmitting fields losslessly, with reduced precision, or not at all. As was discussed earlier, several configurations might be in simultaneous use, by different data users, or for different purposes.

6. Lossless Compression Results and Analysis

There are several criteria by which we may evaluate lossless compression.

6.1 Speed and complexity

The advantages of testing multiple compression and differencing algorithms comes at a considerable cost in speed and complexity. In real-world applications, one must limit the algorithm search, based on the test data sets.

The preliminary prototype is probably slow enough that this is insufficient. Work up to this point has been directed towards a proof of feasibility--that is to produce a sufficient compression factor to justify economic viability. Execution speed was considered a low priority.

The prototype which is closer to an operational version will have to be designed to execute more quickly. That issue is addressed in section 8.

6.2 Compression Factors vs Generic Byte Stream Compressors

Table 1 shows the compression factors (C.F.) obtained in FY 94 and FY 95 on thirteen different packet data streams. They are also compared to those obtained by Unix compress and HA 0.98.
It must be noted that very limited information was provided to us on the packet format and statistics during FY 94. In some instances, incorrect formats had to be assumed. Hence, results during FY 94 were not always as good as might have been hoped. During FY 95 additional information became available, and significant additions and improvements were also made to our hybrid algorithm. Therefore our results are now much more encouraging.

As discussed in section 2, lossless compression factors vary with the input data, and cannot be relied upon completely. Data must be buffered to smooth out compression factor variation. In the worst case, the system must be able to continue after a data loss due to inadequate compression. The fact that no compression system can compress random data is illustrated in the results from Tape BB.

Tape D was of an unexpected format, a type of text with a great deal padding. The best our system could do was to apply HA 0.98 to each block of packets. It did slightly worse than Unix compress, probably because Unix compress was applied to the data set as a whole, rather than to 512 packet blocks, as was done for our system and for HA 0.98 alone. The compression factors achieved for Tape D were quite good, in any event.

No importance is attached to the results for Tape D or Tape BB, since data of these format is not expected to be compressed by our software. In every other case our system outperformed both of the generic byte stream compressors. Furthermore, in every case other than Tape BB we achieved a lossless compression factor of at least 3.14 (in some cases substantially higher)--a very promising result which indicates that our compression algorithm has the potential to substantially reduce transmission and archive costs.

The results presented here suggest that, with some buffering, compression factors of at least 3 can be safely assured for all of the types of data tested in this study, except for the encrypted data set.

Table 1 Lossless Compression Factors

Data Set	Description	# of Packets	Fiscal 94 C.F.	Fiscal 95 C.F.	Unix Comp-ress	HA 0.98
A	Short runs of different packet types.	5000	1.23	4.73	2.04	3.22
		44160	1.25	4.72	1.96	3.16
B	Mostly of expected data format.	5000	2.88	3.27	1.67	2.49
		125000	2.75	3.19	1.68	2.48
C	Mostly of expected data format.	5000	2.74	3.15	1.67	2.48
		19200	2.82	3.28	1.73	2.62
D	Unexpected format: Padded text. Will not actually occur.	5000	7.69	15.23	9.28	15.23
		125000	8.05	15.39	15.76	15.39
G	Not originally of expected format, but is now.	5000	2.44	3.14	1.76	2.04
		125000	2.58	3.32	1.85	2.13
H	Not originally of expected format, but is now.	5000	n/d	3.45	1.72	2.27
		125000	n/d	3.60	2.09	2.16
		375000	n/d	3.25	1.93	2.00
I	Not originally of expected format, but is now.	5000	n/d	3.80	1.85	2.75
		125000	n/d	4.75	2.07	3.25

Data Set	Description	# of Packets	FY 94 C.F.	FY 95 C.F.	Unix Comp-ress	HA 0.98
J	Not originally of expected format, but is now.	5000	n/d	4.14	1.78	2.71
		125000	n/d	3.53	1.66	2.25
K	Not originally of expected format, but is now.	5000	n/d	4.63	1.94	2.89
		125000	n/d	3.83	1.86	2.42
L	Not originally of expected format, but is now.	5000	n/d	4.49	1.70	2.66
		125000	n/d	4.04	1.75	2.52
M	Not originally of expected format, but is now.	5000	n/d	4.85	2.00	3.00
		125000	n/d	4.32	1.90	2.76
AA	Expected format.	2841	3.30	3.54	1.71	2.44
BB	Encrypted data. Impossible for us to compress.	5000	.999	.999	0.72	.999
		8128	n/d	.999	0.76	.999

n/d: Not done. Some data sets were only recently made available. These were not run through the old versions of the software, as there was no need.

6.3 Choice of Optimal Algorithm

In our tests, algorithms 6 (LZ77 Applied to Differences), 7 (LZRW3A) and 8 (LZRW3A Applied to Differences) were never or almost never picked as best, but the rest were at least occasionally picked. The most commonly picked algorithms are 1 (Constant Coding), 4 (Rice Coding), 11 (HA 0.98), 12 (HA 0.98 Applied to Differences), 13 (Radix Coding), 15 (Arithmetic Coding) and 16 (Arithmetic Coding Applied to Differences).

It was not always easy to predict which algorithm would perform best, based on the known data contents. This demonstrates the desirability of testing multiple algorithms. However, most fields generally did best with one or two algorithms.

As an example, Table 2 shows the results from a 14 bit field, from Tape AA, for 6 blocks. The selected algorithm is underlined).

Table 2 Compression factors for a field in which values tend to repeat

Algorithm #																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n/a	1.99	1.68	2.41	2.15	2.16	n/a	n/a	2.27	2.25	2.72	2.70	2.29	2.28	2.65	2.64	2.35
n/a	1.99	1.84	2.24	2.43	2.42	n/a	n/a	2.27	2.26	2.78	2.76	2.17	2.16	2.46	2.45	2.23
n/a	1.74	1.54	2.09	2.30	1.49	1.77	n/a	2.32	1.54	2.88	2.01	1.96	1.94	2.39	1.92	2.06
n/a	1.74	1.27	2.23	2.39	1.62	1.77	n/a	2.30	1.67	2.89	2.18	1.96	1.97	2.36	2.10	2.17
n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

(n/a means could not be applied, or was worse than original):

The first 5 intervals had 512 input packets, the last had 281. In the last two intervals, the field was too noisy for any compression.

The variation in the algorithm 2 compression factor indicates that the number of constant bits varied from interval to interval. Some methods involving first differencing performed well, indicating it is partially valid to model the values as a smooth curve. However, some dictionary search methods also performed well, indicating that the data has a tendency to repeat itself. Where compression was possible, method 11 (HA 0.98) performed best.

Table 3 shows the results for a 4 bit field, from the same data set, on which dictionary search algorithms and HA 0.98 both performed poorly. Algorithm 13 (Radix coding) generally performed best, indicating that the contents are essentially random, but are restricted to some range.

Table 3 Compression factors for a field which is essentially random, but of restricted range

Algorithm #																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n/a	1.99	1.08	2.32	1.70	1.69	n/a	n/a	1.80	1.79	n/a	n/a	2.49	2.46	2.42	2.39	2.14
n/a	1.99	1.10	2.40	1.66	1.65	n/a	n/a	1.84	1.78	n/a	n/a	2.49	2.46	2.43	2.40	2.15
n/a	1.99	1.11	2.49	1.78	1.75	n/a	n/a	1.81	1.77	n/a	n/a	2.49	2.46	2.47	2.44	2.15
n/a	1.99	n/a	2.48	1.75	1.75	n/a	n/a	1.85	1.80	n/a	n/a	2.49	2.46	2.47	2.44	2.20
n/a	1.99	1.16	2.41	1.73	1.72	n/a	n/a	1.78	1.83	n/a	n/a	2.49	2.46	2.43	2.40	2.23
n/a	1.98	1.06	2.32	1.54	1.52	n/a	n/a	1.76	1.71	1.68	1.65	2.46	2.42	2.34	2.30	2.03

Finally, Table 4 shows the results from a 16 bit field, from the same data set. This is a clear case where testing multiple algorithms can substantially improve performance. The best algorithms were 4 (Rice Coding) and 16 (Arithmetic Coding Applied to Differences). Both involve coding the remapped first differences, indicating that this field follows a fairly smooth curve. Intervals in which method 4 performed better presumably have regions of different activity levels, and so benefitted from its small scale adaptivity.

Table 4 Compression for a field with mixed statistics

Algorithm #																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n/a	1.59	2.46	9.15	2.32	6.39	1.07	3.88	1.45	7.46	2.91	8.12	1.92	5.09	1.66	9.48	8.87
n/a	1.33	1.75	7.09	1.68	4.80	n/a	n/a	1.10	5.35	2.07	5.76	1.75	4.42	n/a	6.99	7.05
n/a	1.59	1.94	7.34	1.88	5.25	n/a	n/a	1.19	5.36	2.03	6.19	1.76	5.41	1.61	7.49	7.15
n/a	1.45	1.94	7.45	1.83	5.04	n/a	n/a	1.19	5.56	2.16	5.78	1.77	5.08	n/a	7.49	7.24
n/a	1.59	2.23	7.83	2.05	5.15	n/a	n/a	1.29	5.70	2.33	5.95	1.79	4.82	1.63	7.45	7.59
n/a	1.76	2.15	7.59	2.04	5.63	n/a	n/a	1.35	5.88	2.19	5.94	1.99	5.22	1.78	7.74	7.24

7. Lossy Compression Results and Analysis

The lossless compression factor for Tape AA was 3.54. Purely as a demonstration, TapeAA compression was made lossy by setting iLoss to 8 for one field, to 4 for two fields, and to 16 for one more field, creating maximum de-compression errors of 4, 2 and 8, respectively. The compression factor increased significantly, to 4.58. That represents a reduction in data volume by 23%. The field content is discussed in [2].

When many different groups integrate their efforts into one system, it is quite likely that greater precision is needed for some projects than others. Furthermore, they may feel that adding a few extra significant bits has little impact on the uncompressed data volume. The packet format they agree upon will more or less include enough significant bits to satisfy everyone.

Unfortunately the last few significant bits tend to have the most activity, and to look the most random. That means that those few extra bits may substantially impact compressed data volume, as in this case.

In this system we have only included the option of reducing the precision of the information. Other forms of lossy compression are possible which perform data smoothing. For example, if one were to average pairs of data points together, the number of points transmitted would be reduced, and the results would probably be smoother, further increasing the compression factor.

We are not advocating or imposing lossy compression. Rather, it is our intent to make available a compression system in which any particular data user could elect to make information requests with a custom configuration which reduces the precision of some data fields. This would enable those users that wish to do so to trade off bandwidth vs precision, or to transmit more packets at the same bandwidth by reducing precision.

8. Suggestions for the Development of Future Packet Formats

In section 1.1 we noted that packet data streams represent an administratively efficient solution to the problem of system integration, but that the administrative efficiency is gained at a considerable cost in technical inefficiency:

(1) Higher data volumes and transmission rates. There are many costs associated with these problems. For example, in typical remote sensing systems, higher data rates require more transmit power, resulting in more expensive power systems, and greater payload weight, which has a large impact on system costs and capabilities.

(2) Greater information complexity. This leads to more complex hardware and software to process the data. That increases costs. Most importantly, it greatly increases the probability of hardware and software design errors.

Consequently, a technically inefficient packet format can substantially increase costs, development time, and risk of failure.

This section is intended to provide guidance to the designers of future packet data streams. We have five concrete suggestions:

(1) Keep the format simple. There have been many complex data formats in the scientific and engineering community which have included unnecessary levels of complexity, such as data format descriptors, position pointers, and the like. Data format which only contain the necessary information are much more efficient, and require much less complex processing.

(2) Use bit packed integral binary formats which do not waste any bits. Binary information is inherently more efficient than text or binary coded decimal (BCD), and integers are much easier to transport, and are usually more efficient. If some users might have difficulty using bit packed binary information, the packet specification could easily include sample source code to unpack the data into text form.

(3) Specify large packet sizes, so that there can be many data points per packet. Each packet in a complex system involves a substantial amount of administrative overhead, in terms of sync bits, packet ID's, routing information, telemetry flags, time tags, error detection bits, and the like. Increasing the overall packet size decreases the relative fraction of storage, processing, and transmission associated with the overhead.

Using larger packets simplifies other phases of processing. Small packets lead to the creation of many different packet types to handle the various housekeeping data fields. That increases hardware and software complexity. Furthermore, many modern packet communications systems can modify the order of packet arrival. Hence there must be a packet sort module which places the packets back into order. Larger (and therefore fewer) packets reduce the costs of packet sorting.

(4) Employ error correction coding, error detection, and error recovery, in order to reduce problems associated with faulty data transmission and storage.

(5) Employ data compression systems carefully tailored to the data packet stream format and content, such as the system discussed in this study. The added cost and complexity of the compression system is often well worth the reduction of other costs.

9. Proposal for Future Development

This section is a proposal for future development during FY 96. We now propose to focus on creating a prototype which is more practical. That is, it will be closer to being an operational implementation. The steps that may be worked on include:

(1) Complete algorithm 17 (Arithmetic Coding/Rice hybrid).

(2) The current system is customized by modification of the source code for a few isolated subroutines. This is too complex for a near-operational or operational implementation, especially if the system is implemented in hardware, because configurations may have to change, and it is desirable to allow a choice of multiple possible simultaneous configurations for different projects or data formats.

Therefore the packet format customization information will be moved to external tables. These could be held in files for software implementation, or in separate PROM modules for hardware implementation.

This improvement could also add features such as taking advantage of the restrictions on values implied by packet type classification, and combining certain fields from several packet types into one compression stream.

(3) It was noticed that a large improvement in compression factors occurred after the current system was improved to include all of the frequent types of packet that occurred in the test data sets. Due to development time constraints, this was not always done in the most optimal fashion.

The easier customization may allow a better application to the current packet types. We will also actively seek more test data sets, in order to better compress all of the main packet types.

As a separate task, we may also compress a meteorological database, although it is not clear if the packet compressor represents the best approach to that problem. It may be that off-the-shelf free or commercial software, such as that included with the database package, will provide adequate results.

(4) The current prototype stops if an error occurs. An operational system must continue with minimal loss of information.

(5) Some additional options in lossy data compression should be considered, including at least one (such as averaging) that employs data smoothing.

As stated earlier, lossy compression will not be imposed, but will be a configuration option for those groups wishing to employ it.

(6) By far the most important issue to be dealt will be execution speed. There are several approaches that may be taken:

A. Improvements in source code efficiency.
B. Narrowing the choice of compression algorithms for each field within each packet type.
C. Sacrificing some compression factor in order to use faster algorithms.
D. Possibly, beginning the development of hardware, such as creating specifications for one or more application specific integrated circuits (ASIC).

(7) The current system is implemented as software on Sun SparcStations. This will be expanded to include other computing platforms.

(8) Provisions must be made to handle real-time data streams, such as data timeout and sync.

(9) Substantial testing and validation must be done.

(10) It will be necessary to begin assembling the System Design Requirements for any operational systems. Depending on the application, this typically involves such issues as the choice of

computing platforms and/or custom hardware, space qualification (if appropriate), power, weight and size.

(11) Delivery of Documentation and prototype source code.

9. Conclusion

The methodologies discussed in this study can produce significantly higher compression factors on packet data streams than generic byte stream compression compressors, though at a cost in complexity and speed. It seems likely that data compression can substantially reduce the costs associated with transmitting and archiving packet data streams. With all of the types of data that have been tested here, a compression factor of at least three can usually be assured, provided adequate buffering is used to smooth out the variation. We now propose to transition efforts towards producing a practical operational system.

10. References

[1] Bell, T.C., Cleary, J.G. and Witten, I.H., Text compression, Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Choi and Grunes, Computer Systems, Packet Formats and Test Data Sets for the Lossless Data Compression of Packet Data Streams, October 1995.

[3] Choi and Grunes, Applications of Data Compression Techniques to a Communication System, NRL Memorandum Report 8140.2-95-7742, June, 1995.

[4] Grunes and Choi, "A Packet Data Compressor", NASA GSFC SIMDCWS 95 Proceedings, October, 1995.

[5] Gailly, Jean-loup, "Comp.Compression Frequently Asked Questions", posted to Usenet news group comp.compression, 27 July, 1995.

[6] MIL-STD-973, Configuration Management, AMSC No. D6728, 17 April 1992.

[7] Rice, R.F., "Some Practical Universal Noiseless Coding Techniques", JPL Publication #79-22, NASA, JPL/CIT Pasadena, CA, March, 1979.

[8] Rice, "Some Practical Noiseless Coding Techniques, Part III, Module PSI14,K+", JPL Publication #91-3, NASA, JPL/CIT, Pasadena, CA, November, 1991.

[9] Williams, Adaptive Data Compression, Kluwer Academic Publishers, Boston, MA, 1991, pp 35.

[10] Williams, ibid, pp 37.

[11] Williams, "An extremely fast ZIV-Lempel Data Compression Algorithm", Proc. of Data Compression Conference, IEEE TH0373-1/91, pp 362-371. Actually describes algorithm LZRW1. The LZRW3A algorithm was described in a computer archive (ftp://ftp.adelaide.edu.au/pub/compression).

APPENDIX A
CONFIGURATION MANAGEMENT PLAN

A1. Introduction

A1.1 Task

The main objective of this study has been to develop a compression algorithm that can generate substantial cost savings and bandwidth reduction. The work has included the development of a proof of feasibility prototype compression and de-compression system, which can be configured to a wide variety of packet formats and missions.

This appendix deals with the administrative issues. As such it supplements the technical information provided in Reference [5]. It also supplies some minor details that would not have been appropriate for inclusion in the main body of the report.

The purposes of CM plans are discussed in section 3.24 of [3].

The current prototype system has been implemented in software. In addition to discussing basic configuration management issues, this document will specify the systems onto which the software and test data are loaded, the methods by which the current system is compiled, run, and transported, and the methods by which it has been tested. It also briefly discusses the files comprising the system.

A1.2 This Document

The scope of this document applies to that part of the work performed during FY 94, FY 95, and proposed to occur during FY 96. It principally defines the details of the work associated with the system as it currently stands.

The format and content of this plan are intended to comply as much as is practical with MIL-STD-973. However, those aspects of MIL-STD-973 which are very labor intensive or which require action on the part of the sponsor have been omitted.

Development of this configuration plan will be incremental. The software is still preliminary, not operational. The interfaces, working groups, and operational procedures relating to an operational version cannot be determined. What can be described at the current time are current status and near-term development plans.

A1.3 Computers

A need exists for a system which can be used with both proprietary and non-proprietary format data. Therefore, software development and some initial testing with released data is done at a non-proprietary site on two systems from which files can be transported without excessive security precautions. These systems are herein called System1 and System2. The editing of proprietary information specification files, and final testing, are performed at a proprietary site on three systems containing the proprietary information--herein called System3, System4 and System5, at a facility called EWN.

All the mentioned systems are various models of Sun Sparc work station, running under various Sun Unix operating systems. It is probable that the software could be compiled and run on some other work

stations, but that has not been verified. If this system moves closer to being a practical prototype, versions will have to be produced that function on other (TBD) types of computer system.

Most of the non-proprietary development work was done on System1, a Sun SparcStation 1. This computer contains a full copy (except for the proprietary information) of the current prototype version (Rev 0.36), and of the source code for many prior versions.

Executables for the current version are in directory ~grunes/choi and source code for the current version is in directory ~grunes/choi/Rev0.36.

Other versions are archived in other sub-directories of ~grunes/choi/Rev* (see section 7 for a list).

(~<user> is a convention understood by Unix systems to indicate the login directory of <user>.)

The older versions have been compressed using Unix Compress, in order to save space. One may uncompress a version by moving into its directory and typing

```
uncompress *
```

Note that Unix Compress has no relationship to the compression software that this document deals with--it was used to compress those files because it is widely available, although it often does not achieve very high compression factors.

The information on System1 in ~grunes and its sub-directories is occasionally backed up to the NRL file archiver, called Nrlfs1.

System1 is rather slow. Therefore System2, a Sun SparcStation 20 located at the same facility, is used for preliminary testing of the prototype software, on the two files (TapeAA and TapeBB) which were approved for use at the non-proprietary site.

All of the test data, including TapeAA and TapeBB, and the current version of the software, are present at the EWN facility, in a Sun SparcStation 10 called System3, again in directories ~grunes/choi and ~grunes/choi/Rev0.36. Two of the source files, otherclass.h and othercl2.h (in both directories), were changed at that site only, in order to reflect proprietary information about packet formats. One report, in rep9509a.doc and rep9509b.doc, is not present at the non-proprietary site. System3 shares file systems with a system called System4, which is used to load the software from floppy disk, and to load the test data from tape. It also shares file systems with a system called System5, which may be used instead of System4 to load the software from floppy disk.

A1.4 Abbreviations

The following abbreviations are used within this document:

ATSC	AlliedSignal Technical Services Corporation
CM	Configuration Management
FY	Fiscal Year
GNU	GNU, Not Unix (A Label Associated with the Free Software Foundation)
IBM	International Business Machines, Inc.
NRL	Naval Research Laboratory
MTI	Mentor Technologies, Inc.

A2. Reference Documents

The reference numbers in these appendix correspond to those in the main body of the report. This plan has been created in a form which is intended to follow most of the guidelines of [6]. The interim technical report for the Data Compression task produced at the end of FY 94 was eventually released as an NRL Memorandum Report, [3]. The data formats involved are discussed in [2].

A3. Organization

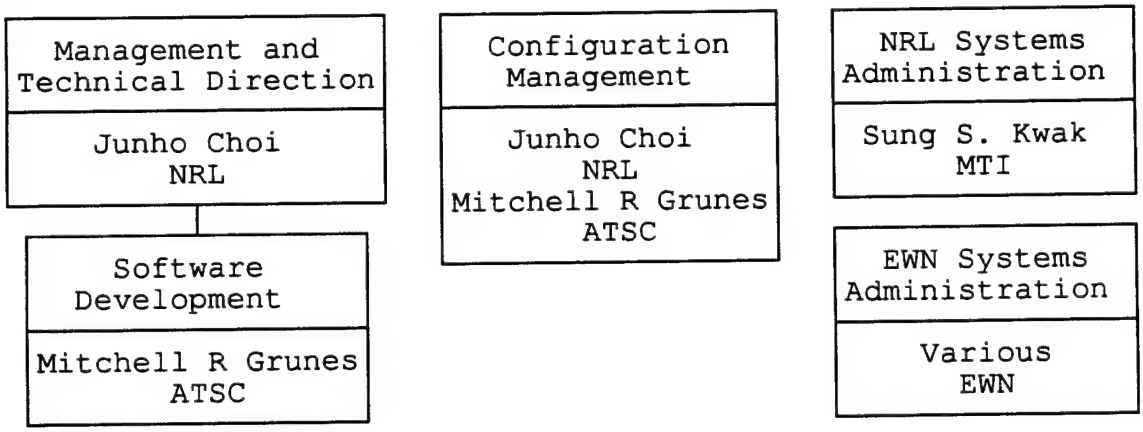
The primary goal up to this point has been to prove the feasibility of using data compression to cut operating costs. The compression factors achieved appear to be sufficient to prove this feasibility.

For the purposes of this document, the functions within this task are herein designated as Management and Technical Direction, Software Development, System Administration, and Configuration Management. Because of the preliminary nature of this work, and the small number of people involved, the organizational aspects of this task have been minimal, and these functions overlap in terms of personnel.

Overall Management and Technical Direction for the Data Compression task is provided by Junho Choi (NRL). Software Development has been provided by Mitchell R Grunes (ATSC).

Software Development has had to interface directly with the various Systems Administrators. For example, for security reasons, direct System Administrator involvement is required to loading information into EWN facility (see section 5).

Configuration Management is also provided by Junho Choi and Mitchell R Grunes.



Some changes may occur during FY 96. For example, it appears that development of an optimal compression algorithm requires sample data and bit field boundary information for all of the data types expected to be processed. This will require our task to actively seek out such information by contacts with appropriate parties.

It has not been determined whether the operational product should be hardware, software, or some combination of both. If hardware is involved, interface with some Hardware Development group may be required.

In addition, testing and verification may be required by other parties in order to improve quality assurance.

A4. Configuration Management Phasing and Milestones

We herein describe and portray the sequence of events and milestones for the implementation of Configuration Management in phase with major program milestones and events.

Milestones were previously established for FY 95:

Milestones from FY 95	O	N	D	J	F	M	A	M	J	J	A	S
Complete Algorithm 17 (Arithmetic Coding/Rice hybrid		X	X									
Reduce Overhead Bits												
Improve Predicts				X	X							
Improve Rice Coding		X	X			X			X			
Radix Coding						X						
Arithmetic Coding							X	X	X			
LZ78				X	X							
Simple Lossy Compression										X	X	X
Compress Meteorological database								X	X	X		
Deliver: Prototype Source Code												X
Deliver: Documentation												X

An unexpected delay occurred because the security managers of the EWN facility imposed the reasonable but unanticipated requirement that the components taken from the externally developed software packages (LZRW3A and HA 0.98) be carefully examined to ensure that they could not deliberately or accidentally sabotage anything. As a result the compression of the meteorological databases did not occur in this fiscal year, and the documentation was slightly delayed. All of the other components were completed as planned.

During FY 96 we propose to focus our efforts on developing a prototype system for practical application. These plans are discussed in greater detail in section 8 of the main report. These plans may be modified during development or by the sponsor, but proposed milestones are:

Milestones for FY 96	O	N	D	J	F	M	A	M	J	J	A	S
Complete Arithmetic Coding/Rice Hybrid		X										
Table driven packet customization		X	X									
Obtain more packet types, experiment with customization						X	X	X	X	X	X	
Compress Meteorological database		X	X	X	X							
Improve error handling				X	X							
Further work on Lossy Compression									X			
Improve execution speed: Source code efficiency Narrowing algorithm choices Compromise algorithms Hardware development?					X X	X X X	X X X	X X X	X X X	X X X ?	X X ?	
Other computing platforms						X	X					
Real time data stream handling							X	X				
Test and Validation								X	X			
System Design Requirements									X	X		
Deliver: Prototype Source Code												X
Documentation	X										X	X

A5. Data Management

Most of the data sets used for testing have been provided in the form of Exabyte (8 mm) tapes, or of 9 track tapes.

Two data sets (TapeAA and TapeBB) have been released for use at the non-proprietary facility. The remaining test data sets in current use are considered proprietary, and transport outside the EWN facility must be done with the involvement of the ADP Security officer of the EWN facility. The reading of software into the EWN facility must also be done with the involvement of the EWN Systems Administrators, or the ADP Security Officer.

The current protocol is to retain all of the data sets used for testing for the duration of the development task.

During the process of personnel changeover, the media containing two of the test data sets (TapeE and TapeF) were re-used for other purposes. No records were kept as to their data content, and they cannot

be re-constructed. This presents no technical problem, as it is relatively easy to obtain additional test data. It is unfortunate because the technical report ([3]) produced at the end of FY 94 mentioned TapeE. No technical reports ever have or ever will make reference to TapeF, which was used only once to verify that some procedure was working.

As an additional precaution to safeguard the data in the future, the first 125000 packets of each available test data set are now stored on System3 (in Unix Compressed form), which is itself backed-up on a regular basis. Information about the test data sets is logged into file

~grunes/choi/tapekey

on computer System3.

A6. Configuration Identification

A6.1 Test Data Set Identification

No changes are expected to be made to the current test data sets, although additional test data sets may be added. Therefore no configuration version numbers need to be assigned to them. They are referred to in this document as:

TapeA
TapeB
TapeC
TapeD
TapeG
TapeH
TapeH.big
TapeI
TapeJ
TapeK
TapeL
TapeM
TapeAA
TapeBB

All these data sets are archived on System3 in directory ~grunes/choi, as compressed by Unix Compress (e.g., TapeA was compressed to ~grunes/choi/TapeA.Z).

Additional information on these data sets is available in [2].

A6.2 Software Identification

The software has gone through multiple revisions, and will go through others. The overall compression algorithm is a hybrid of many simpler algorithms. The revisions have partly been for the purpose of adding or improving the compression algorithms in the hybrid, and partly for the purpose of better adapting the software to the format and statistics of the data. At preliminary implementation, these improvements have sometimes introduced temporary defects into the software. Some revisions have been designed to remove the defects in other revisions. In addition, some revisions have changed the information output by the software. The current revision level (0.36) has no known defects, and has been subjected to a moderately high degree of testing with the current test data sets (see section 7.4).

No log has been kept of the changes that were made in each revision, owing to the preliminary and non-operational state of the prototype. However, archives of prior versions are being kept on System1. Each revision has been assigned a version number. These version numbers all start with "0.", indicating that they are preliminary. The first released version would start with "1.". On System1, the source code for each version is in a separate directory. Past and current versions include:

~grunes/choi/Rev0.0
~grunes/choi/Rev0.1
~grunes/choi/Rev0.2
~grunes/choi/Rev0.21
~grunes/choi/Rev0.22
~grunes/choi/Rev0.23
~grunes/choi/Rev0.24
~grunes/choi/Rev0.25
~grunes/choi/Rev0.25.long (variant of Rev0.25 which did not terminate after first few packets)
~grunes/choi/Rev0.26
~grunes/choi/Rev0.27
~grunes/choi/Rev0.28
~grunes/choi/Rev0.29
~grunes/choi/Rev0.30
~grunes/choi/Rev0.31
~grunes/choi/Rev0.32
~grunes/choi/Rev0.33
~grunes/choi/Rev0.34
~grunes/choi/Rev0.35
~grunes/choi/Rev0.36
~grunes/choi/Rev0.36.Lossy (Version of Rev0.36 which performed lossy compression on type 3 packets. In particular, SpecPak was modified for type 3 by setting iLoss to 8 for field 12, to 4 for fields 14 and 16, and to 16 for field 19.)

If this prototype is adapted for operational use, it will be desirable to keep logs of the changes that occur in each version, and the reasons for those changes. A future version of this document could formalize those requirements, if further development occurs on this task.

A6.3 Descriptions of Files

This section describes the files comprising the compression/de-compression system at the current time.

The following file name suffixes were used:

File name suffix	File category
.c	C language source code
.for	Fortran language source code
.h	C or Fortran source code included by some other program(s)
.sh	Unix command sequences. May be executed by source <filename>

.sc — "Scratch" files:
Temporary files
Output from the last set of test runs

.Z — A file compressed by Unix Compress. May be de-compressed by
uncompress <filename>
The uncompressed file name would be identical, but without the .Z. For example, ~grunes/choi/Rev0.34/callpdc.for.Z is a Unix Compressed version of the fortran source callpdc, for an old revision.

(no suffix) — Other:
Executables--programs which have been compiled and linked, and are ready to run. The name of the executable is the prefix of the file name which contains the mainline program.

Data files--various data files.

In alphabetical order, the following files are located in ~grunes/choi. Some scratch files may not always be present. Additional documentation is included in the source code itself.

a.sc — Scratch file holding test run output.

a.sc.TapeA.Z
a.sc.TapeB.Z
etc. — Unix compressed scratch file holding test run the output from test runs for TapeA, TapeB, etc. See section 6.1.

b.sc — Temporary file holding output from individual program. Unix commands head, tail and grep are used during processing to extract portions of this into a.sc.

blockha — Executable derived from the latest revision blockha.for and other files. The same compression software as callpdc, but limited to the HA 0.98 algorithm (method 11), working on blocks the blocks of 512 packets as byte streams (not as separate fields).

callpdc — Executable derived from the latest revision callpdc.for and other files. The compression program.

callpdd — Executable derived from the latest revision callpdd.for and other files. The de-compression program.

comp.sc — Scratch file holding output from callpdc or blockha--the compressed data. Serves as input for callpdd.

input.sc — Scratch file holding input for callpdc or blockha--the original data.

mitchcopy — Executable derived from the latest revision mitchcopy.c. The disk/tape file copy program.

output.sc — Scratch file holding output from callpdd--the de-compressed data.

rep9509.doc This is a draft of the main report
Choi and Grunes, Lossless Data Compression of Packet Data Streams.
intended for use by the Technical Information Division of NRL to produce an NRL Memorandum Report. It was prepared using Word Perfect 5.1 for DOS.

rep9509a.doc Draft of the supplementary report containing proprietary information:
rep9509b.doc Choi and Grunes, Computer Systems, Packet Formats and Test Data Sets for the Lossless Data Compression of Packet Data Streams.
Not present in the System1 archive. It was prepared using Microsoft Word 5.1 for Macintosh format. rep9509b.doc duplicates some rep9509a.doc, but puts them properly in landscape orientation.

runall.sh Runs the test sequence rundisk.sh on all of the data sets, and renames the output to a.sc.TapeA, a.sc.TapeB, etc. This file is only present on System3.

rundisk.sh Procedure copied from the latest revision source code directory. Runs the test sequence on a data set which has been copied to file Tape. Output goes to a.sc.

runtape.sh Procedure copied from the latest revision source code directory. Creates file Tape from a tape drive, then runs rundisk.sh. Purely an example.

Rev0.0
Rev0.1
etc. Directories holding revisions of source code. See section 6.2

TapeA.Z
TapeB.Z
etc. (Unix compressed) test data sets.

tapecf Log used to record the compression factors extracted from the run output files. Only present on System3.

tapekey Log containing descriptions of the test data sets. Only present on System3.

unpack.sh Procedure copied from the latest revision source code directory. Used to unpack the archive extracted from the floppy disk, when moving files from System1 to System3.

The following files are found in ~grunes/choi/Rev0.36.

acoder.c Part of the HA 0.98 software package, somewhat modified by Software Development. Contains the arithmetic coding routines used by HA 0.98.

acoder.h Part of the HA 0.98 software package.

asc.c Part of the HA 0.98 software package, somewhat modified by Software Development. Associated with the HA 0.98 ASC compression method.

asc.h Part of the HA 0.98 software package.

blockha.for Modified version of callpdc.for used to create blockha. This is a variant of the mainline program which calls the compression software.

bufio.c Stubs to allow Fortran programs (including those in mitchlib.for) to call Unix low level I/O routines. Used by the BUFIO section of mitchlib.for. Contains source code for:

Routine	Fortran Call	Is a stub to call
openc_	call OpenC(iUnit,FilNamAr,InOut)	open or creat (open or create file)
closec_	call CloseC(iUnit)	close (Close file)
readblock_	call ReadBlock(iUnit,nByte,A)	read (Read data)
writeblock_	call WriteBlock(iUnit,nByte,A)	write (Write data)
seekc_	call SeekC(iUnit,iPos)	lseek (Seek file)

bufio.h Include file used by mitchlib.for to specify parameters and common variables for buffered unformatted I/O.

callpdc.for Mainline program for compression. This is the mainline program which calls the compression software.

callpdd.for This is the mainline program which calls the de-compression software. Also includes source code for:

subroutine PutData(iData,iPakType,Index)	Subroutine to output or store de-compressed data.
subroutine PutData2(FileData)	Subroutine to output 1 record.
function igetc_comp(idummy)	Reads one byte from compressed stream.
function iRdbit(nBit)	Read iVal from Pack buffer in nBit bits.
logical function GetFlag(iVal,nBit, idefault)	Read condition bit. If true, read iVal, else=default.

check.h Sets icheck to 1 to indicate that output from de-compression will be checked against original input.sc file.
This must be changed to 0 for lossy compression, or if check is not desired.

compile.sh Procedure to compile programs.

compress.h Part of the LZRW3-A package.

debug.h Indicates desired level (normally 1) of debug output.

error.c Part of the HA 0.98 software package, somewhat modified by Software Development.

error.h	Part of the HA 0.98 software package, somewhat modified by Software Development.
fast_copy.h	Part of the LZRW3-A package.
ha.h	Part of the HA 0.98 software package, somewhat modified by Software Development.
haio.h	Part of the HA 0.98 software package, somewhat modified by Software Development.
hsc.c	Part of the HA 0.98 software package, somewhat modified by Software Development.
hsc.h	Part of the HA 0.98 software package, somewhat modified by Software Development.
lzrw3-a.c	Part of the LZRW3-A package, somewhat modified by Software Development. The external interface routine is lzrw3a_, called from Fortran as call lzrw3a(1,nInput,input,nOutput,output)
machine.h	Part of the HA 0.98 software package, somewhat modified by Software Development.
maketar.sh	Procedure to generate a compressed tar archive, and copy some executables and procedures to the directory above.
mask.h	Contains bit masks, used by many programs.
mitch_ha.c	Contains Fortran callable stub to call the HA 0.98 software package. Called from Fortran as call mitch_ha(MitchInLen,MitchIn,iWhich,iDirection,MaxMitchOutLen, MitchOutLen,MitchOut)
mitchcopy.c	A disk and tape file copying utility that does not suffer from the block size restrictions of Unix dd and copy commands. Unlike Unix tcopy, it only copies one file, and allows one to limit file size. A mainline program.
mitchlib.for	Contains some general library routines, used by compression and de-compression programs. Includes the following packages and routines:

BIGINT: Simple multiple precision non-negative integer arithmetic package, to do addition, decrement, multiplication and division.

---

subroutine BigTo(i,n, a)	Convert normal non-negative integer i to big format.
function iBigFm(n,a)	Convert big format to normal integer.
subroutine BigCopy(n,a, b)	Copy b to a.
function LoadByte(a)	Convert byte to unsigned integer.

function iGetnUse(n,a)	Get number of bytes coded in a(n) and a(n-1).
subroutine PutnUse(nUse,n, a)	Stuff nUse into a(n) and a(n-1).
subroutine BigAddI(n,a,i)	Big format mixed addition.
subroutine BigAdd(n,a,b)	Big format integer addition.
subroutine BigDecr(n,a)	Big format integer decrement.
subroutine BigMulAddI(n,a,i,j)	Big format mixed multiplication and addition.
subroutine BigMul(n,a,b, c)	Big format integer multiplication.
subroutine BigDivI(n,a,i, iRem)	Big format mixed division.
subroutine BigPrint(n,a,mes)	Print a big number.

BUFIO Fast raw sequential or direct buffered file I/O.

---

block data bd_bufio	Defines common data variables.
subroutine OpenIn(iUnit,FilNam)	Open file for input.
subroutine OpenOut(iUnit,FilNam)	Open file for output.
subroutine CloseFile(iUnit)	Close file on unit iUnit, flushing buffers and initializing variables.
subroutine ReadFile(iUnit,nByte,A,*)	Function to Read nByte bytes from unit iUnit to byte array A.
iReadFile1(iUnit)	Read 1 unsigned byte from unit iUnit.
subroutine WriteFile(iUnit,nByte,A)	Write nByte bytes from byte array to unit iUnit.
subroutine FlushFile(iUnit)	Flush file buffers.
subroutine ReadBlock(iUnit,nByte,A)	Read nByte bytes of data into A.
subroutine WriteBlock(iUnit,nByte,A)	Write nByte bytes of data from A.
subroutine SeekFile(iUnit,iPos)	Seek (position) file to 1-origin byte position iPos.

==========

MISCELLANEOUS Other Routines

	subroutine PrintStr(iUnit,string,iflush)	If iflush=0, append string into line, print if full. If iflush>0, flush.
	subroutine PrintAr(iUnit,label,iArray,n)	Print array.
	subroutine PrintCnt(iUnit,label,Iarray,n)	Print counts of occurrences.
	subroutine Shell(command)	Execute operating system command.
	integer*2 function int2(i)	Function missing from Sun.
	subroutine Err(iflag,a)	An error message handler.
	subroutine WrBit(iVal,nBit)	Write iVal in nBit bits.
	subroutine InitWrBit	Initialize WrBit buffers.
	subroutine FlushWrBit	Flush WrBit buffers.
nrlpdc.for	Most of the subroutines specific to the compression program. Included routines are:	
	subroutine NRLPDC(iflush,iData, iPakType,nField,nSubField,nBit, iBlock,Lossless,iLoss,MaxStore,iStore,nPack)	Main compression subroutine.
	subroutine BitPlusFollow(iVal, iBitsToFollow,iUse,iTry)	Arithmetic coding routine to output 1 bit, plus opposite values.
	function CF(nBitsOrig,nBits)	Compression factor.
	subroutine FitDelta(iDelta,nStore,iUse, MinDiff,MaxDiff,iAlpha,iBeta)	Try to improve predicts using least squares fit.
	function iGCF(iUse,iStore,nStore, MaxVal1, MaxVal2,iTotal)	Find, send and divide by greatest common factor.
	subroutine ToByte1(nStore,iStore, nBit,MaxnBuf, Buf,iBuf)	Part 1/2 in byte stream handler.
	subroutine ToByte2(iUse,nStore, nBit,nBuf,Buf,MaxnBuf, iTry)	Part 2/2 in byte stream handler.
	function iReMap1(iVal,MinAbs, MinVal,MaxVal,iBest,SomeNeg)	Reversibly remap value iVal into non-negative number.
	function iSendRice(iUse,iDiff,k,mx2,	Send one Rice code.

nFill,iWork)	
function iSendRice2(iUse,iStore,J,k, MaxRem,nFill,iWork)	Rice Code a Sequence.
function iTry0(iUse,iStore,nStore, nBit)	Try Method 0=Uncompressed.
function iTry1(iUse,iStore,nStore,nBit, iConstant)	Try Method 1: Constant value.
function iTry3(iUse,iStore,nStore,nBit, MaxRun,nBitRun)	Try Method 3: Run Length Encoding.
function iTry4(iUse,iStore,nStore, iBlock,MaxRem,iRice, kArray, kDelta,iWork)	Try Method 4: Rice.
function iTry5(iUse,iStore,nStore,nBit, LinkList)	LZ77G technique.
function iTry7(iUse,iStore,nStore, nBit)	LZRW3a technique.
function iTry9(iUse,iStore,nStore,nBit, LinkList,ii1,iiL)	Apply LZ78G technique.
function iTry11(iUse,iStore,nStore, nBit)	Try Method 11: HA.
function iTry13(iUse,iStore,nStore, nBit)	Try Method 13: Radix Coding.
function iTry15(iUse,iStore,nStore,nBit)	Try Method 15: Arithmetic Coding.
function iTry20(iUse,iStore,nStore, nBit,Lossless,	Try Method 20: RLE+Rice.
function iTryReMapAll(iStore,nStore, nBit,iUse,NoRepeat, Lossless,MapStyle,iDelta,iWork, MaxRem,iAlpha,iBeta)	Try each MapStyle for ReMap.
function iWhichRice(iStore,i1,J,nBit)	Choose which Rice code to use.
subroutine PrintID(nPack,iPID)	Print included packet ID fields
subroutine ReMap(iStore,nStore, MapStyle,nBit,iUse,NoRepeat,	Turn values into differences from predicts and remap to a positive

	Lossless,iAlpha,iBeta,iDelta,iWork, iTry,MaxRem,iSum)	number.
	subroutine RemCnBits(iStore,nStore, nBit, mask,iConst,nBitKeep)	Remove Constant bits.
	logical function SendFlag(cond,iVal, nBit,iUse,iTotal)	If cond false, send 0 bit. If cond true, send 1 bit, and send value.
	subroutine TryAll(iField,iBlock, Lossless,iStore,nStore,nBit,nBitOrig, MinBits,iDelta,iWork1,iWork2,iWork3)	Try all compression methods.
	function iWrRad(iVal,MaxVal,flush,iUse)	Write iVal using radix coding.
nrlpdc.h	Include file, used by callpdc.for, nrlpdc.for, and specpak.for.	
nrlpdd.for	Most of the subroutines specific to the de-compression program. Included routines are:	
	subroutine NRLPDD(iData, iPakType, nBit,MaxStore,iStore,nPack,PutData)	Main de-compression routine.
	subroutine FmByte1(nStore,nBit,nBuf, Buf, iStore)	Undoes ToByte1.
	subroutine FmByte2(nStore,nBit, MaxnBuf, nBuf,Buf)	Undoes ToByte2.
	subroutine GetRice2(iStore,J,k, MaxRem,iRice, nFill,iWork)	Get a Rice Sequence.
	subroutine GetUnMapInfo(nBit, nStore,MapStyle10,Lossless, MapStyle,NoRepeat,iGCF1,iGCF2, iPredict1,MinVal,MaxVal5, MaxRem,MinAbs,SomeNeg,UseFit, iAlpha,iBeta)	Read info required to unmap differences.
	function iGetRice(k,mx2)	Get one Rice code.
	function iUnMap1(iReMap,MinAbs, MinVal,MaxVal,iBest,SomeNeg)	Reverse iReMap1.
	subroutine RdRad(nVal,MaxValAr, iValAr)	Read iValAr using radix coding--undoes iWrRad.
	subroutine UnMap(iStore,nStore,nBit, MapStyle,iGCF1,iGCF2,MinVal, MinAbs,MaxVal5,MaxRem,SomeNeg,	Unmap remapped values.

UseFit,iAlpha,iBeta)	
subroutine UnTry0(iStore,nStore,nBit)	Uncompress Method 0: No compression.
subroutine UnTry1(iStore,nStore,nBit)	Uncompress Method 1.
subroutine UnTry3(iStore,nStore,nBit)	Uncompress Method 3.
subroutine UnTry4(iStore,nStore, iBlock,MaxRem,kArray)	Uncompress Method 4.
subroutine UnTry5(iStore,nStore,nBit)	Uncompress Method 5.
subroutine UnTry7(iStore,nStore,nBit)	Uncompress Method 7.
subroutine UnTry9(iStore,nStore,nBit, ii1,iiL)	Uncompress Method 9.
subroutine UnTry11(iStore,nStore, nBit)	Uncompress Method 11.
subroutine UnTry13(iStore,nStore, nBit,iWork)	Uncompress Method 13.
subroutine UnTry15(iStore,nStore, nBit)	Uncompress Method 15.
subroutine UnTry20(iStore,nStore, nBit,Lossless, iWork1,iWork2)	Uncompress Method 20.
subroutine UnTryAll(iStore,nStore, nBit,Lossless, iBlock,iWork1,iWork2)	Uncompress all methods.

nrlpdd.h — Include file, used by callpdd.for and nrlpdd.for.

otherclass.h — Include file, used to classify certain types of packets on the basis of proprietary packet ID information. The version included in the System1 archive is ineffectual. The version in System3 reflects the proprietary information.

othercl2.h — Include file, used to determine which of the proprietary packet types should have their data word components treated as a byte stream (vs as separate words). The version included in the System1 archive merely selects packet type 2, which includes packets of the correct generic type, but which have packet ID's which the software does not specifically address. The version in System3 reflects proprietary information. It was determined by empirical testing on each of the proprietary packet types.

port.h — Part of the LZRW3-A package.

rdbit.h — Include file, used to hold common block variables associated with bit field unpacking routine irdbit.

rundisk.sh — Procedure used to run the test sequence on file Tape. Output is saved to file a.sc.

runtape.sh — Procedure used to run the test sequence on all the test data sets.

specpak.for — Contains routines common to the compression and de-compression software. This differs from mitchlib, in that the routines are not generic enough to be likely to be included in other software packages.

Some of these routines are very specific to the packet format. Specifically, routine Classify determines packet types, SpecPak provides the compression and de-compression software with information pertaining to the data fields, UnPak unpacks the original packets into the data fields, and Pak packs the data fields back into data packets.

Included routines are:

subroutine Classify(FileData,nPack, iPakType)	Determine packet types.
subroutine ContractBits(mask, MaskBits)	Contract 24 bit word.
subroutine ExpndBits(mask,MaskBits)	Expand 24 bit word.
function nBitNeed(i)	# of bits to send i.
subroutine InsCnBits(iStore,nStore, nBit, mask,iConst)	Insert Constant bits.
subroutine Pak(iData,iPakType, FileData)	Pack packets.
subroutine SpecPak(iPakType,nField, nSubField,nBit,iBlock,LossLess, iLoss)	Specify packet formats.
subroutine UnPak(FileData,iPakType, iData)	Unpack data.
subroutine ApplyFit(iDiffOld,iAlpha, iBeta,MinD,MaxD,MinDiff,MaxDiff, Wrap, iDiffBest)	Revise predicted difference using parameters from iFitDelta.
subroutine WrapVal(iVal,Min,Max)	Wrap ival around if outside range.

swdict.c — Part of the HA 0.98 software package, somewhat modified by Software Development.

swdict.h — Part of the HA 0.98 software package, somewhat modified by Software Development.

unpack.sh Procedure used to unpack the contents of the compressed tar archive, after moving the archive from System1 to System4.

wrbit.h Include file, used to hold common block variables associated with bit field packing routine wrbit.

A7. Interface Management

A7.1 General Information

The interface between the user and the software, and between the software and the data it inputs and outputs, are both in preliminary state. This is necessary because it has not yet been determined what form (e.g., tape, disk, network lines, etc.) the data will be input and output in. The interface is sufficient for initial testing and development, but is not adequate for an operational product. This version of plan documents the current configuration.

A7.2 Compilation of Software

Most of the source code developed by Software Development is in the Fortran 77 programming language, using a few common extensions. However, it proved most practical to perform unformatted I/O (input and output) in C. Two externally developed software packages (LZRW3-A and HA 0.98) were also written in C.

The current software version has been successfully compiled and run on Sun Sparc work stations System1, System2 and System3.

System1 is running SunOS release 4.1.3. Software is compiled on System1 under release 2.0.1 of the Sun F77 Fortran compiler and release 2.6 of the GNU GCC compiler.

System2 is running SUNOS release 5.4. Software is compiled and run on System2 under release 3.01 of the Sun F77 Fortran compiler and either the /usr/ucb/cc compiler, or the SunPro C compiler.

System3 is running SunOS release 5.3 (Unix System 5 Release 4.0). Software is compiled on System3 under release 2.0.1 of the Sun F77 Fortran compiler and release 2.01 of the Sun CC compiler. For security reasons, and to take into account the modified otherclass.h and othercl2.h files, the software is re-compiled on System3. (As of this writing the system administrators of System3 are in the process of upgrading System3 to SunOS release 5.4.)

Sun operating systems and C language compilers vary substantially from version to version. Very careful software development is needed to ensure that C language programs will compile under more than one version. Our Software Development group attempted to create portable code. Unfortunately, that could not be required of the externally developed software packages. Some of the externally developed C software will not compile under an early version of the Sun CC compiler, which is loaded on System1 (so GCC was used instead). It cannot be guaranteed that the software will continue to compile, link and run under future Sun software releases.

It is necessary to match the versions of the Sun Fortran and C compilers, or object modules may not link properly. However, executables created on earlier versions of the operating system seem to run properly on later versions of the operating system. For example, executables created on System1 run on System2).

Compilation is fairly simple on System1 and System3. Log into user account grunes, and type:

```
cd choi/Rev0.36
source compile.sh
```

This runs the compilers, and creates the executables. Some warnings are given during this procedure:

(1) "mv" commands are used to rename some files. If renaming is not needed, these commands print a warning. Renaming was used because source code has often been edited on an IBM PC compatible computer running under MS-DOS, which could not handle long file names.

(2) The C compiler and linker may give additional warnings. On System1, the warnings look like

```
ld.so: warning: /usr/lib/libc.so.1.8.1 has older revision than expected 9
```

On System3 the warnings resemble

```
semantics of "<operation>" change in ANSI C; use explicit cast
```

All these warnings may be safely ignored.

One may compile the software on System2 in the same way as on System3, with a few more warning messages.

A7.3 Transport of Software

Development occurs on System1, which is backed up on the NRL file archiver to prevent loss of data. Software is then transferred to System2 and System3. The System2 archive may be deleted to save space.

The grunes account on System1 is backed up to the NRL file archiver by logging into System1 account grunes and typing:

```
tar -cvf system1.tar * .??*
compress system1.tar
ftp nrlfs1
grunes
(enter password)
binary
send system1.tar.Z
quit
rm system1.tar.Z
```

To transfer the executable software from System1 to System2, Log onto System2 account grunes and type:

```
cd ~grunes/choi
mkdir Rev0.36
ftp system1
grunes
(enter password)
binary
prompt
mget *.sh
get callpdc
get callpdd
```

```
get blockha
get mitchcopy
cd Rev0.36
lcd Rev0.36
mget *
bye
```

Modify the compilation procedure to reflect the use of a different compiler, by replacing "gcc" with "cc -I. -Xc":

```
cd ~grunes/choi/Rev0.36
vi compile.sh
:%s/gcc/cc -I. -Xc/
:wq
```

The software is transported from System1 to System3 on a Sun Unix tar format 3.5" DSHD (double sided high density) floppy disk. Before creating that disk, compile the software on System1 (see previous section), so that all files will be renamed and in the proper location.

The floppy disk should be formatted by placing it into a 1.44 MB drive on an IBM PC compatible computer, and typing

```
format b: /u
```

Sun work stations can also format floppies, but have problems doing so under certain circumstances.

On System1, type:

```
cd ~grunes/choi/Rev0.36
source maketar.sh
```

This deletes all of the object and executable files, creates a compressed archive (~grunes/choi/nrlpdc.tar.Z), and copies four files (mitchcopy, unpack.sh, rundisk.sh, and runtape.sh) into ~grunes/choi.

Finally insert the formatted floppy disk into System1, and type:

```
tar -cvf /dev/fd0a nrlpdc.tar.Z unpack.sh
eject floppy
rm nrlpdc.tar.Z
```

To load the software into System3, one could log into System4 user account grunes, load the floppy and type:

```
cd ~grunes/choi
volcheck
tar -xvf /vol/dev/rdiskette0/unlabeled
eject
```

However, the system administrators of System3, System4, and System5 have insisted that they load the software themselves, for security reasons. They sometimes prefer to use their own procedures to do so.

It is strongly advised that they be told to do the equivalent of:

```
su -
cd ~grunes/choi
volcheck
tar -xvf /vol/dev/rdiskette0/unlabeled
eject
chown grunes *
```

(For some reason it may be necessary to give everyone write permission to ~grunes and ~grunes/choi before root can write to those directories.) If they choose to use another procedure, make sure that they copy the contents into directory ~grunes/choi. Leaving out the "volcheck" could cause the somewhat unreliable floppy drive to fail. Leaving out the "chown" could cause problems with future updates, and make it impossible to delete the archive file.

Regardless of how the software is loaded, one should complete the load under account grunes by typing

```
cd ~grunes/choi
source unpack.sh
```

Use the modified otherclass.h and othercl2.h files by typing

```
cd ~grunes/choi
cp otherclass.h Rev0.36
cp othercl2.h Rev0.36
```

If otherclass.h should ever be modified to add packet type numbers greater than 20, one must also alter MaxiPakType in nrlpdc.h and nrlpdd.h.

Modify the compilation procedure to reflect the use of a different compiler, by replacing "gcc" with "cc -I. -Xc":

```
cd ~grunes/choi/Rev0.36
vi compile.sh
:%s/gcc/cc -I. -Xc/
:wq
```

Compile the software as specified in the prior section.

A7.4 Running the Software

At the current time, the prototype software takes its data from disk files, and places its output into disk files. The test data was loaded to disk from magnetic tape cartridges and 9 track tapes. Those tapes contained a header file, the data file, and a trailer file.

Several different techniques were tested to load the data. Initially, attempts were made to use the following Unix utilities to copy the data:

```
dd
cpy
tcopy
```

Unix dd and cpy failed due to restrictions on tape block sizes. The Unix tcopy command tries to copy all of the files, cannot be told to stop after a certain file size, and produces an error message. Therefore use of all of those utilities was abandoned.

The administrators of System3 and System4 have also developed software which can make tape-to-disk copies. However, tests indicated that it produces a copy which does not appear to be in the appropriate format. (It might be that it adds a header section which causes the packet boundaries to be misaligned. The compression software is designed to work on the packet data, not the header and trailer files. The packet data is located on the second file on those tapes.) The administrators have since then stated that it might also be able to produce an original format copy, but this has not been verified at this time.

The simplest solution, for the present, is to use a copy program called mitchcopy, developed for other purposes. To load the data onto disk, log into System4 user account grunes, load the tape into tape drive /dev/nrst2 (other tape drives can also be used) and type:

```
cd choi
mt -f /dev/nrst2 rewind
mt -f /dev/nrst2 fsf 1
mitchcopy /dev/nrst2 TapeZ 2000000
```

The output file name TapeZ is arbitrary. The 2000000 limits the disk file size. To save disk space, apply Unix Compress:

```
compress -f TapeZ
```

which outputs a smaller archive called TapeZ.Z.

If new files to be tested are added, one should edit file

```
~grunes/choi/runall.sh
```

to include tests on the new data sets.

One can run all of the software and tests (discussed in more detail in sections 7.5 and 7.6) by typing

```
cd ~grunes/choi
nice -20 source runall.sh &
```

The "nice -20" helps ensure that this long job will run at reduced priority. The "&" causes the program to run as a "batch" job, which means that it will continue running after one logs off. (Prior to logging off, results will also be output to the screen.)

After the procedure is complete, there will exist, in directory ~grunes/choi, output files

```
a.sc.TapeA
a.sc.TapeB
 etc.
```

which can be examined. The output files can be compressed using

```
cd ~grunes/choi
compress a.sc*
```

The procedure runall.sh actually invokes another procedure, rundisk.sh, for each of the data sets. If one only wants to run the procedure for one data set, type

```
cd ~grunes/choi
zcat TapeZ.Z > Tape
nice -20 source rundisk.sh &
```

(where TapeZ.Z is the name of the Unix compressed data set).

Runall.sh outputs the results of the software and tests to file a.sc.

A7.5 Testing and Verification

This is a preliminary prototype, not operational software, and it is not necessary that the software be free of all defects. Nonetheless, no defects are known to exist in the current version (0.36) of the software, which was tested on all of the test data sets.

The compile.sh procedure compiled the software with error checking (such as Fortran language subscript checks and floating point exceptions) turned on, in order to improve the probabilities that an error would be detected. In all of the tests, all of the programs ran to completion, indicating that these problems did not exist.

The test procedure (runall.sh) involved running all of the following tests, on each of the test data sets:

1. The first 80000 bytes of the file is copied to file input.sc.

2. Unix compress is tried on this file. The compression factor for Unix compress can be computed as the ratio of the sizes of file input.sc and the compressed file input.sc.Z. Unix uncompress is used to restore the file to a readable state.

3. The compression executable, callpdc, is run to produce compressed file comp.sc. It prints out various statistics. At the end it prints out a final cumulative compression factor. In every instance the program ran to completion without error.

4. The compression executable, callpdd, is run to produce de-compressed file output.sc. The program is currently set to compare each output packet to the corresponding packet in input.sc, and fail if there is a difference. In every instance the program ran to completion without error.

5. The Unix cmp command is used to test whether input.sc and output.sc are identical. This test was successful in every instance.

6. Steps 2-5 were repeated on the whole data set. In every instance, all tests were successful.

All of these tests were performed by Software Development. As the prototype moves closer to being an operational version, other parties should become involved in software testing and verification.

A7.6 Sample Output

This section lists the lines of the rundisk.sh procedure, and the output that it produces, using a sample run with TapeAA as an example.

To run this, login to the grunes account and type

```
cd ~grunes/choi
zcat TapeAA > Tape
nice -20 source rundisk.sh
```

The first few lines of rundisk.sh remove "aliases" that system administrators sometimes use to modify the actions of Unix commands, and delete old scratch files:

```
#--------------------RUNDISK.SH-----------------------
# Run this using
#      cp <diskfile> Tape
#      source rundisk.sh
# This will take a long time.

echo ------------------------------------------------
echo Deleting aliases and old files.
     unalias ls
     unalias mv
     unalias rm
echo rm Tape.Z core \*.sc input.sc.Z Rev0\*/\*.sc
     rm Tape.Z core  *.sc input.sc.Z Rev0*/*.sc
echo ------------------------------------------------ |& tee -a a.sc
echo ---------This will take a long time---------    |& tee -a a.sc
echo ---------You may as well walk away----------    |& tee -a a.sc
sleep 5
```

The above lines will output the following to the screen and to the a.sc file:

```
------------------------------------------------
---------This will take a long time---------
---------You may as well walk away----------
```

Rundisk.sh then lists the date and the characteristics of the Tape file:

```
echo date                                            |& tee -a a.sc
     date                                            |& tee -a a.sc
echo ls -lad Tape                                    |& tee -a a.sc
     ls -lad Tape                                    |& tee -a a.sc
```

which produces the following output:

```
date
Tue Sep 19 15:10:54 EDT 1995
ls -lad Tape
-rw-r--r--  1 grunes      45456 Sep 19 15:09 Tape
```

Rundisk.sh then runs Unix compress on a short run (limited to at most 80000 bytes=5000 packets):

```
echo ------------------Short Run------------------      |& tee -a a.sc
echo ./mitchcopy Tape input.sc 80000 v=0                |& tee -a a.sc
     ./mitchcopy Tape input.sc 80000 v=0                |& tee -a a.sc

echo ------------------------------------------------ |& tee -a a.sc
echo --------Try Unix Compress---------                |& tee -a a.sc
echo ls -lad input.sc                                  |& tee -a a.sc
     ls -lad input.sc                                  |& tee -a a.sc
echo nice -20 compress -f input.sc                     |& tee -a a.sc
     nice -20 compress -f input.sc                     |& tee -a a.sc
echo ls -lad input.sc.Z                                |& tee -a a.sc
     ls -lad input.sc.Z                                |& tee -a a.sc
echo nice -20 uncompress input.sc                      |& tee -a a.sc
     nice -20 uncompress input.sc                      |& tee -a a.sc
```

which produces the following output:

```
------------------Short Run-----------------
./mitchcopy Tape input.sc 80000 v=0
src=Tape dest=input.sc m=80000 b=65535 s=0 v=0
rec 1  Input bytes=45456  Output bytes=45456  Errors=       0
------------------------------------------------
```

```
--------Try Unix Compress---------
ls -lad input.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:10 input.sc
nice -20 compress -f input.sc
ls -lad input.sc.Z
-rw-r--r--  1 grunes      26518 Sep 19 15:10 input.sc.Z
nice -20 uncompress input.sc
```

One may compute the compression factor as 45456/26518=1.71.

Rundisk.sh then runs the compression software on the short sample, but only lists the last 13 lines of its output. "Nice" is used to reduce execution priority:

```
echo ------------------------------------------------- |& tee -a a.sc
echo --------Try NRLPDC with all methods---------      |& tee -a a.sc
echo nice -20 ./callpdc                                |& tee -a a.sc
     nice -20 ./callpdc                                |& tee b.sc
echo  ----------Last 13 output lines----------         >> a.sc
     tail -13 b.sc                                     >> a.sc
```

which produces the following output:

```
-------------------------------------------------
--------Try NRLPDC with all methods---------
nice -20 ./callpdc
----------Last 13 output lines----------
           ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
 19    2     1.22  4496 *****  3672  3957  3775  3960  4043 ***** *****
             5620  5683 ***** *****  3698  3778 ***** *****  3777  3797
 PIDs=281*23
   281 Input Packets   Packet Type=  3 CF=    4.03636 rec=  2561
 CF=orig_bits/compressed=  35968/  8944=    4.02147 with overhead
 Cumulative CF          =  363648/  102792=    3.53771
 ==End of File==
 ==  0 Warnings         0 Errors==
 Note: this program was linked with -fast or -fnonstd
 and so may have produced nonstandard floating-point results.
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

The only important components here are

```
Cumulative CF          =  363648/  102792=    3.53771
```

which shows that the compression factor was about 3.54, and

```
==  0 Warnings         0 Errors==
```

which shows that the software completed without error.

The warning about nonstandard floating-point results only means that the program would have aborted if a floating point error had occurred, which was considered a desirable goal while testing the prototype. The warning can be safely ignored.

Rundisk.sh then runs the de-compression software, and compares the original uncompressed input to the de-compressed output:

```
echo nice -20 ./callpdd                                |& tee -a a.sc
     nice -20 ./callpdd                                |& tee b.sc
```

```
echo ----------Last 10 output lines                    >> a.sc
     tail -10 b.sc                                     >> a.sc

echo ------------------Doing Compare------------------  |& tee -a a.sc
echo cmp input.sc output.sc                            |& tee -a a.sc
     cmp input.sc output.sc                            |& tee -a a.sc
echo ls -lad \*.sc                                     |& tee -a a.sc
     ls -lad  *.sc                                     |& tee -a a.sc
```

which produces the following output

```
nice -20 ./callpdd
----------Last 10 output lines
 Packet Type,# decompressed packets=  3  512
  Packets     2049-    2560 bytes    32769-   40960
 Packet Type,# decompressed packets=  3  281
  Packets     2561-    2841 bytes    40961-   45456
 ==End of File==
 ==  0 Warnings          0 Errors==
 Note: this program was linked with -fast or -fnonstd
 and so may have produced nonstandard floating-point results.
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
------------------Doing Compare------------------
cmp input.sc output.sc
ls -lad *.sc
-rw-r--r--  1 grunes       2124 Sep 19 15:12 a.sc
-rw-r--r--  1 grunes        890 Sep 19 15:12 b.sc
-rw-r--r--  1 grunes      12849 Sep 19 15:11 comp.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:10 input.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:12 output.sc
```

The fact that no error message appears after the

```
cmp input.sc output.sc
```

command is a good sign. Rundisk.sh then runs BLOCKHA on the same data, and once again decompresses the result and runs a comparison, then deletes scratch files:

```
echo --------Try BLOCKHA---------                     |& tee -a a.sc
echo nice -20 ./blockha                                |& tee -a a.sc
     nice -20 ./blockha                                |& tee b.sc
echo  ----------Last 13 output lines----------         >> a.sc
     tail -13 b.sc                                     >> a.sc

echo nice -20 ./callpdd                                |& tee -a a.sc
     nice -20 ./callpdd                                |& tee b.sc
echo ----------Last 10 output lines                    >> a.sc
     tail -10 b.sc                                     >> a.sc

echo ------------------Doing Compare------------------  |& tee -a a.sc
echo cmp input.sc output.sc                            |& tee -a a.sc
     cmp input.sc output.sc                            |& tee -a a.sc
echo ls -lad \*.sc                                     |& tee -a a.sc
     ls -lad  *.sc                                     |& tee -a a.sc
echo rm b.sc input.sc comp.sc output.sc                |& tee -a a.sc
     rm b.sc input.sc comp.sc output.sc                |& tee -a a.sc
```

which produces the following output:

```
--------Try BLOCKHA---------
nice -20 ./blockha
----------Last 13 output lines----------
            ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

  1   11     2.52 35968 ***** 35976 39342 35133 21355 21224 20269 20267
            18326 18340 14286 14300 35992 35998 24455 24464 35234 33681
   281 Input Packets   Packet Type=  1 CF=    2.51683 rec=  2561
 CF=orig_bits/compressed=  35968/  14328=    2.51033 with overhead
 Cumulative CF          =  363648/  148848=    2.44308
 ==End of File==
 ==  0 Warnings         0 Errors==
 Note: this program was linked with -fast or -fnonstd
 and so may have produced nonstandard floating-point results.
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
nice -20 ./callpdd
----------Last 10 output lines
 Packet Type,# decompressed packets=  1  512
  Packets     2049-    2560 bytes    32769-   40960
 Packet Type,# decompressed packets=  1  281
  Packets     2561-    2841 bytes    40961-   45456
 ==End of File==
 ==  0 Warnings         0 Errors==
 Note: this program was linked with -fast or -fnonstd
 and so may have produced nonstandard floating-point results.
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
------------------Doing Compare-----------------
cmp input.sc output.sc
ls -lad *.sc
-rw-r--r--  1 grunes       3726 Sep 19 15:13 a.sc
-rw-r--r--  1 grunes        890 Sep 19 15:13 b.sc
-rw-r--r--  1 grunes      18606 Sep 19 15:13 comp.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:10 input.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:13 output.sc
rm b.sc input.sc comp.sc output.sc
```

Rundisk.sh then runs Unix compress on the entire test data set:

```
echo ------------------Long Run-----------------       |& tee -a a.sc
echo ./mitchcopy Tape input.sc v=0                     |& tee -a a.sc
     ./mitchcopy Tape input.sc v=0                     |& tee -a a.sc

echo ------------------------------------------------- |& tee -a a.sc
echo --------Try Unix Compress---------                |& tee -a a.sc

echo ls -lad input.sc                                  |& tee -a a.sc
     ls -lad input.sc                                  |& tee -a a.sc
echo nice -20 compress -f input.sc                     |& tee -a a.sc
     nice -20 compress -f input.sc                     |& tee -a a.sc
echo ls -lad input.sc.Z                                |& tee -a a.sc
     ls -lad input.sc.Z                                |& tee -a a.sc
echo nice -20 uncompress input.sc                      |& tee -a a.sc
     nice -20 uncompress input.sc                      |& tee -a a.sc
```

which produces the following output:

```
------------------Long Run-----------------
./mitchcopy Tape input.sc v=0
src=Tape dest=input.sc m=none b=65535 s=0 v=0
```

```
rec 1  Input bytes=45456  Output bytes=45456  Errors=       0
-------------------------------------------------
--------Try Unix Compress---------
ls -lad input.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:13 input.sc
nice -20 compress -f input.sc
ls -lad input.sc.Z
-rw-r--r--  1 grunes      26518 Sep 19 15:13 input.sc.Z
nice -20 uncompress input.sc
```

Rundisk.sh then runs the compression software. This time it includes the first 800 lines of output:

```
echo ------------------------------------------------- |& tee -a a.sc
echo --------Try NRLPDC with all methods---------      |& tee -a a.sc
echo nice -20 ./callpdc                                |& tee -a a.sc
     nice -20 ./callpdc                                |& tee b.sc
echo ----------First 800 output lines----------       >> a.sc
     head -800 b.sc                                    >> a.sc
echo "----------Last 13 output lines, if any"         >> a.sc
     tail +800 b.sc |& tail -13                        >> a.sc
```

The beginning of the output looks like:

```
-------------------------------------------------
--------Try NRLPDC with all methods---------
nice -20 ./callpdc
----------First 800 output lines----------
          ----------CallPDC Rev 0.36 9/14/95----------

                    -----------------Bits For Each Method-----------------
                      0     1     2     3     4     5     6     7     8
 Fld Best             Orig  Const Const RLE   RiceG LZ77G LZ77G LZRW3 LZRW3
  #  Meth   CF                     BitRm       +Diff       +Diff   A   +Diff
 --  ---- ------- ----- ----- ----- ----- ----- ----- ----- ----- -----

                9     10    11    12    13    14    15    16    17    20
               LZ78G  HA    HA    Radix Radix Arith Arith Rice  RiceG
                      +Diff       +Diff       +Diff       +Diff +Arit +RLE
               ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

  1    1   512.00  1024     2 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
  2    1   512.00   512     1 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
  3   13     2.49  2048 ***** 1030  1895   883  1202  1211 ***** *****
             1140  1143 ***** *****  822   832   846   856   864   955
  4    1   512.00  4608     9 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
  5    1   512.00  5120    10 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
  6    1   512.00  8192    16 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
  7   16     9.48  8192 ***** 5142  3330   895  3531  1282  7632  2112
             5650  1098  2817 1009  4271  1611  4928   864   872   924
  8    1   512.00  1536     3 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
  9    1   512.00  1536     3 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
 10    1   512.00  1536     3 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
 11    1   512.00  1024     2 ***** ***** ***** ***** ***** ***** *****
            ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
 12    4     1.26  5632 ***** 5643  6359  4484  5093  5672  7125  8108
             5244  6631  4582 5493  4910  4970 ***** 4930  4582  4570
```

```
13    1   512.00  1024     2 ***** ***** ***** ***** ***** ***** *****
           ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
14   15     1.76  3072 *****  2056  3078  1911  2676  2680 ***** *****
           2365  2370  1993  1999  1913  1923  1744  1751  1877  2037
15    1   512.00  1536     3 ***** ***** ***** ***** ***** ***** *****
           ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
16   15     1.26  2560 *****  2565  3529  2237  3096  3121  4070  4084
           2749  2753  2223  2237  2504  2521  2034  2055  2244  2405
17    1   512.00  1024     2 ***** ***** ***** ***** ***** ***** *****
           ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
18   11     2.72  7168 *****  3605  4275  2975  3312  3322 ***** *****
           3161  3181  2639  2651  3132  3146  2703  2717  3035  3051
19    2     1.23  8192 *****  6675  7191  6771  7194  7263 ***** *****
          10469 10335  7686  7690  6701  6772 ***** *****  6837  6802
PIDs=512*23
  512 Input Packets   Packet Type=  3 CF=    3.37588 rec=  1
CF=orig_bits/compressed=  65536/  19448=    3.36981 with overhead
Cumulative CF          =  65536/  19448=    3.36981
```

The column labeled "Fld #" shows the field #. In this packet format (3) there are 19 fields. "Best Meth" contains the method number which performed best on that block of 512 packets for that field. "CF" indicates the compression factor for that field, but does not include all applicable forms of overhead. The remaining columns show the number of bits that each method required to represent the sequence. Note that there were 19 methods listed (including Method 0 = no compression), and the fact that there is no method 18 or 19. In addition method 17 (Rice-like adaptivity with arithmetic coding) is listed, but is never used, because the de-compression component of the software has not yet been written for method 17. (In some instances a warning would be given relating to method 17. This warning would list the value of "MinBits", which is the best possible value excluding method 17, and iiTry17, which is the value for method 17. This warning is of no importance, but only indicates that method 17 would sometimes have improved matters.) These fields take two lines to print.

Some of the columns contain "*****". This is because of one of the following reasons:

1. The field is constant. In this event method 1 (Constant Coding) always performs best, so it would be a waste of time to look at the other methods.

2. One of the methods, as currently written, would not work. This might be because the software has a limit on the bit field width, or because the method outputs more bits than the original data stream for this block. Method 1 (constant coding) only works when the field is constant.

In the above example, method 1 (Constant Coding) performed best on field 1. Ignoring overhead, method 0 (the original data stream for field 1) required 1024 bits (2 bits/packet), but constant coding only required 2. This gives a compression factor of 1024 / 2 = 512.00.

Similarly, method 13 (Radix Coding) performed best on field 3. Ignoring overhead, Method 0 required 2048 bits (4 bits/packet), but method 13 only required 822. This gives a compression factor of 2048 / 822 = 2.49.

After the table one sees

```
PIDs=512*23
  512 Input Packets   Packet Type=  3 CF=    3.37588 rec=  1
```

which indicates that there were 512 packets with packet ID value 23, that were grouped together as packet type 3. The compression factor for that packet type within the first block was 3.37588. The block of packets started at record 1.

That compression factor does not include the overhead for the output compressed packet, or for the specification of packet types. Therefore one also sees

```
CF=orig_bits/compressed=  65536/  19448=    3.36981 with overhead
```

Finally one sees the cumulative CF for blocks of packets up to this point:

```
Cumulative CF          =  65536/  19448=    3.36981
```

This is a simple example. It is possible for a block of packets to contain packets of several types. In that case one would see the tables for each type, but the last two lines only after the last type.

The output continues for additional blocks of packets, ending with:

```
18    1   281.00  3934    14 ***** ***** ***** ***** ***** ***** *****
           ***** ***** ***** ***** ***** ***** ***** ***** ***** *****
19    2     1.22  4496 *****  3672  3957  3775  3960  4043 ***** *****
           5620  5683 ***** *****  3698  3778 ***** *****  3777  3797
PIDs=281*23
  281 Input Packets   Packet Type=  3 CF=    4.03636 rec=  2561
CF=orig_bits/compressed=  35968/  8944=    4.02147 with overhead
Cumulative CF          =  363648/  102792=    3.53771
==End of File==
==  0 Warnings         0 Errors==
Note: this program was linked with -fast or -fnonstd
and so may have produced nonstandard floating-point results.
Sun's implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
```

for the last block. The final cumulative compression factor was 3.53771, or about 3.54.

Rundisk.sh then lists all of the PID lines, compression factors, errors and warnings:

```
echo  ----------PIDs----------                                  >> a.sc
      grep -i pid b.sc                                         >> a.sc
echo  ----------Packet Compression factors----------           >> a.sc
      grep "CF=" b.sc                                          >> a.sc
echo  ----------Errors and Warnings----------                  >> a.sc
      grep "==" b.sc                                           >> a.sc
```

which produces the following output:

```
----------PIDs----------
 PIDs=512*23
 PIDs=512*23
 PIDs=512*23
 PIDs=512*23
 PIDs=512*23
 PIDs=281*23
----------Packet Compression factors----------
   512 Input Packets   Packet Type=  3 CF=    3.37588 rec=  1
 CF=orig_bits/compressed=  65536/  19448=    3.36981 with overhead
   512 Input Packets   Packet Type=  3 CF=    3.37640 rec=  513
 CF=orig_bits/compressed=  65536/  19448=    3.36981 with overhead
   512 Input Packets   Packet Type=  3 CF=    3.45782 rec=  1025
```

```
 CF=orig_bits/compressed=  65536/  18992=    3.45072 with overhead
   512 Input Packets   Packet Type=  3 CF=    3.47930 rec=  1537
 CF=orig_bits/compressed=  65536/  18872=    3.47266 with overhead
   512 Input Packets   Packet Type=  3 CF=    3.84330 rec=  2049
 CF=orig_bits/compressed=  65536/  17088=    3.83521 with overhead
   281 Input Packets   Packet Type=  3 CF=    4.03636 rec=  2561
 CF=orig_bits/compressed=  35968/  8944=    4.02147 with overhead
----------Errors and Warnings----------
 ==End of File==
 ==  0 Warnings         0 Errors==
```

Rundisk.sh once again de-compresses the result and compares the de-compressed data set to the original:

```
echo nice -20 ./callpdd                                          |& tee -a a.sc
     nice -20 ./callpdd                                          |& tee b.sc
echo ----------Last 10 output lines                             >> a.sc
     tail -10 b.sc                                               >> a.sc

echo ------------------Doing Compare------------------           |& tee -a a.sc
echo cmp input.sc output.sc                                      |& tee -a a.sc
     cmp input.sc output.sc                                      |& tee -a a.sc
echo ls -lad \*.sc                                               |& tee -a a.sc
     ls -lad  *.sc                                               |& tee -a a.sc
```

producing the following output:

```
nice -20 ./callpdd
----------Last 10 output lines
 Packet Type,# decompressed packets=  3  512
  Packets     2049-    2560 bytes    32769-   40960
 Packet Type,# decompressed packets=  3  281
  Packets     2561-    2841 bytes    40961-   45456
 ==End of File==
 ==  0 Warnings         0 Errors==
 Note: this program was linked with -fast or -fnonstd
 and so may have produced nonstandard floating-point results.
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
------------------Doing Compare------------------
cmp input.sc output.sc
ls -lad *.sc
-rw-r--r--  1 grunes      28215 Sep 19 15:14 a.sc
-rw-r--r--  1 grunes        890 Sep 19 15:14 b.sc
-rw-r--r--  1 grunes      12849 Sep 19 15:14 comp.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:13 input.sc
-rw-r--r--  1 grunes      45456 Sep 19 15:14 output.sc
```

Once again BLOCKHA is used in a similar fashion:

```
echo ------------------------------------------------- |& tee -a a.sc
echo --------Try BLOCKHA---------                      |& tee -a a.sc
echo nice -20 ./blockha                                |& tee -a a.sc
     nice -20 ./blockha                                |& tee b.sc
echo  ----------Last 13 output lines---------          >> a.sc
     tail -13 b.sc                                     >> a.sc
echo  ----------Packet Compression factors----------   >> a.sc
     grep "CF=" b.sc                                   >> a.sc

echo nice -20 ./callpdd                                |& tee -a a.sc
     nice -20 ./callpdd                                |& tee b.sc
echo ----------Last 10 output lines                     >> a.sc
     tail -10 b.sc                                     >> a.sc
```

```
echo ------------------Doing Compare------------------  |& tee -a a.sc
echo cmp input.sc output.sc                             |& tee -a a.sc
     cmp input.sc output.sc                             |& tee -a a.sc
echo ls -lad \*.sc                                      |& tee -a a.sc
     ls -lad  *.sc                                      |& tee -a a.sc
```

We omit the output for brevity. Rundisk.sh ends with deletion of scratch files, and a listing of date and time:

```
echo rm b.sc input.sc comp.sc output.sc                 |& tee -a a.sc
     rm b.sc input.sc comp.sc output.sc                 |& tee -a a.sc
echo date                                               |& tee -a a.sc
     date                                               |& tee -a a.sc
```

which produces the following output:

```
rm b.sc input.sc comp.sc output.sc
date
Tue Sep 19 15:16:32 EDT 1995
```

A8. Configuration Control

If development of an operational version becomes desirable, a formal system will have to exist for processing Engineering Change Proposals, and for other modifications. Administrative responsibility should rest with

Junho Choi
Code 8140.2
Naval Research Laboratory
Washington DC, 20375
Phone 202-767-9050 (may change), 202-767-9792
FAX 202-767-1317

A9. Configuration Status Accounting

At the current time no plans have been made for configuration status accounting, due to the lack of a formal requirement or budget for configuration management on this task.

A10. Configuration Audits

At the current time no plans have been made for configuration audits, due to the lack of a formal requirement or budget for configuration management on this task.

A11. Subcontractor/Vendor Control

At the current time no subcontractors or vendors are involved. However, if the need for such becomes apparent, an appropriate CM plan will be developed, tailored from the directives in [6].